

Functional Programming with F#

Overview and Basic Concepts

Radu Nicolescu
Department of Computer Science
University of Auckland

27 Sep 2017

- 1 Background
- 2 Overview
- 3 Type System and Type Inference
- 4 Functions
- 5 Lambdas
- 6 printfn
- 7 Immutable Values and Mutable Variables
- 8 Pattern Matching
- 9 Collections
- 10 Recursion or Loops?
- 11 Exercise
- 12 Infinite virtual sequences

Outline

- 1 Background
- 2 Overview
- 3 Type System and Type Inference
- 4 Functions
- 5 Lambdas
- 6 printfn
- 7 Immutable Values and Mutable Variables
- 8 Pattern Matching
- 9 Collections
- 10 Recursion or Loops?
- 11 Exercise
- 12 Infinite virtual sequences

Brief historical overview—1.2

- Alonzo Church's λ -expressions (1930s) ☺
- John McCarthy's Lisp ("List processing") : the 2nd oldest programming language (1958)
 - Scheme (1975)
 - Common Lisp (1984)
 - Clojure (2007)
- Peter Landin's ISWIM ("If you See What I Mean") : statically typed + type inference (1966)
 - Haskell : purely functional (1990)
 - Robin Milner's ML ("metalanguage") : functional + side effects (1973)

Brief historical overview—1.2

- Alonzo Church's λ -expressions (1930s) ☺
- John McCarthy's Lisp ("List processing") : the 2nd oldest programming language (1958)
 - Scheme (1975)
 - Common Lisp (1984)
 - Clojure (2007)
- Peter Landin's ISWIM ("If you See What I Mean") : statically typed + type inference (1966)
 - Haskell : purely functional (1990)
 - Robin Milner's ML ("metalanguage") : functional + side effects (1973)

Brief historical overview—1.2

- Alonzo Church's λ -expressions (1930s) ☺
- John McCarthy's Lisp ("List processing") : the 2nd oldest programming language (1958)
 - Scheme (1975)
 - Common Lisp (1984)
 - Clojure (2007)
- Peter Landin's ISWIM ("If you See What I Mean") : statically typed + type inference (1966)
 - Haskell : purely functional (1990)
 - Robin Milner's ML ("metalanguage") : functional + side effects (1973)

Brief historical overview—1.2

- Alonzo Church's λ -expressions (1930s) ☺
- John McCarthy's Lisp ("List processing") : the 2nd oldest programming language (1958)
 - Scheme (1975)
 - Common Lisp (1984)
 - Clojure (2007)
- Peter Landin's ISWIM ("If you See What I Mean") : statically typed + type inference (1966)
 - Haskell : purely functional (1990)
 - Robin Milner's ML ("metalanguage") : functional + side effects (1973)

Brief historical overview—1.2

- Alonzo Church's λ -expressions (1930s) ☺
- John McCarthy's Lisp ("List processing") : the 2nd oldest programming language (1958)
 - Scheme (1975)
 - Common Lisp (1984)
 - Clojure (2007)
- Peter Landin's ISWIM ("If you See What I Mean") : statically typed + type inference (1966)
 - Haskell : purely functional (1990)
 - Robin Milner's ML ("metalanguage") : functional + side effects (1973)

Brief historical overview—2.2

- Robin Milner's ML ("metalanguage") : functional + side effects (1973)
 - Standard ML (1990)
 - CAML ("Categorical Abstract Machine Language") (1985)
 - OCAML (Objective CAML) : CAML + object layer (1996)
 - Don Syme's F# : OCAML + .NET integration (2005)

Brief historical overview—2.2

- Robin Milner's ML ("metalanguage") : functional + side effects (1973)
 - Standard ML (1990)
 - CAML ("Categorical Abstract Machine Language") (1985)
 - OCAML (Objective CAML) : CAML + object layer (1996)
 - Don Syme's F# : OCAML + .NET integration (2005)

Brief historical overview—2.2

- Robin Milner's ML ("metalanguage") : functional + side effects (1973)
 - Standard ML (1990)
 - CAML ("Categorical Abstract Machine Language") (1985)
 - OCAML (Objective CAML) : CAML + object layer (1996)
 - Don Syme's F# : OCAML + .NET integration (2005)

Brief historical overview—2.2

- Robin Milner's ML ("metalanguage") : functional + side effects (1973)
 - Standard ML (1990)
 - CAML ("Categorical Abstract Machine Language") (1985)
 - OCAML (Objective CAML) : CAML + object layer (1996)
 - Don Syme's F# : OCAML + .NET integration (2005)

Brief historical overview—2.2

- Robin Milner's ML ("metalanguage") : functional + side effects (1973)
 - Standard ML (1990)
 - CAML ("Categorical Abstract Machine Language") (1985)
 - OCAML (Objective CAML) : CAML + object layer (1996)
 - Don Syme's F# : OCAML + .NET integration (2005)

F# FREE OPEN-SOURCE MULTI-PLATFORM

F# : OCAML + .NET = functional programming + statically typed + type inference + side effects + objects + .NET + extensions (actors)

- 1 Functional programming (Functional-first)
- 2 Object-oriented programming
- 3 Language-oriented programming (cf., DSL)

Contrast with Anders Hejlsberg's C#

- 1 Object-oriented programming (Objects-first)
- 2 Elements of functional programming

F# FREE OPEN-SOURCE MULTI-PLATFORM

F# : OCAML + .NET = functional programming + statically typed + type inference + side effects + objects + .NET + extensions (actors)

- 1 Functional programming (Functional-first)
- 2 Object-oriented programming
- 3 Language-oriented programming (cf., DSL)

Contrast with Anders Hejlsberg's C#

- 1 Object-oriented programming (Objects-first)
- 2 Elements of functional programming

F# FREE OPEN-SOURCE MULTI-PLATFORM

F# : OCAML + .NET = functional programming + statically typed + type inference + side effects + objects + .NET + extensions (actors)

- 1 Functional programming (Functional-first)
- 2 Object-oriented programming
- 3 Language-oriented programming (cf., DSL)

Contrast with Anders Hejlsberg's C#

- 1 Object-oriented programming (Objects-first)
- 2 Elements of functional programming

F# FREE OPEN-SOURCE MULTI-PLATFORM

F# : OCAML + .NET = functional programming + statically typed + type inference + side effects + objects + .NET + extensions (actors)

- 1 Functional programming (Functional-first)
- 2 Object-oriented programming
- 3 Language-oriented programming (cf., DSL)

Contrast with Anders Hejlsberg's C#

- 1 Object-oriented programming (Objects-first)
- 2 Elements of functional programming

F# FREE OPEN-SOURCE MULTI-PLATFORM

F# : OCAML + .NET = functional programming + statically typed + type inference + side effects + objects + .NET + extensions (actors)

- 1 Functional programming (Functional-first)
- 2 Object-oriented programming
- 3 Language-oriented programming (cf., DSL)

Contrast with Anders Hejlsberg's C#

- 1 Object-oriented programming (Objects-first)
- 2 Elements of functional programming

F# FREE OPEN-SOURCE MULTI-PLATFORM

F# : OCAML + .NET = functional programming + statically typed + type inference + side effects + objects + .NET + extensions (actors)

- 1 Functional programming (Functional-first)
- 2 Object-oriented programming
- 3 Language-oriented programming (cf., DSL)

Contrast with Anders Hejlsberg's C#

- 1 Object-oriented programming (Objects-first)
- 2 Elements of functional programming

F# FREE OPEN-SOURCE MULTI-PLATFORM

F# : OCAML + .NET = functional programming + statically typed + type inference + side effects + objects + .NET + extensions (actors)

- 1 Functional programming (Functional-first)
- 2 Object-oriented programming
- 3 Language-oriented programming (cf., DSL)

Contrast with Anders Hejlsberg's C#

- 1 Object-oriented programming (Objects-first)
- 2 Elements of functional programming

Outline

- ① Background
- ② Overview
- ③ Type System and Type Inference
- ④ Functions
- ⑤ Lambdas
- ⑥ printfn
- ⑦ Immutable Values and Mutable Variables
- ⑧ Pattern Matching
- ⑨ Collections
- ⑩ Recursion or Loops?
- ⑪ Exercise
- ⑫ Infinite virtual sequences

How to use F# on Windows

- **fsc** : command-line compiler
- **fsi** : interactive
- **VS F#** project
 - uses **fsc**
 - also **fsi**: selection + Alt-Enter
- **LINQPAD**
 - uses **fsc**

How to use F# on Windows

- **fsc** : command-line compiler
- **fsi** : interactive
- VS F# project
 - uses fsc
 - also fsi: selection + Alt-Enter
- LINQPAD
 - uses fsc

How to use F# on Windows

- **fsc** : command-line compiler
- **fsi** : interactive
- **VS** F# project
 - uses **fsc**
 - also fsi: selection + Alt-Enter
- **LINQPAD**
 - uses fsc

How to use F# on Windows

- **fsc** : command-line compiler
- **fsi** : interactive
- **VS** F# project
 - uses **fsc**
 - also **fsi**: selection + Alt-Enter
- **LINQPAD**
 - uses **fsc**

How to use F# on Windows

- **fsc** : command-line compiler
- **fsi** : interactive
- **VS** F# project
 - uses **fsc**
 - also **fsi**: selection + Alt-Enter
- **LINQPAD**
 - uses **fsc**

Resources

- **F# Software Foundation** <http://fsharp.org/>
- **Visual F#** on MSDN <https://msdn.microsoft.com/en-us/library/dd233154.aspx>
- **F# for Fun and Profit**
<http://fsharpforfunandprofit.com>
- **Try F#** <http://www./tryfsharp.org>
- **F# Programming**
http://en.wikibooks.org/wiki/F_Sharp_Programming
 - This handout's sequel is largely based on the "Basic Concepts" section of this wiki book

Resources

- **F# Software Foundation** <http://fsharp.org/>
- **Visual F#** on MSDN <https://msdn.microsoft.com/en-us/library/dd233154.aspx>
- **F# for Fun and Profit**
<http://fsharpforfunandprofit.com>
- **Try F#** <http://www./tryfsharp.org>
- **F# Programming**
http://en.wikibooks.org/wiki/F_Sharp_Programming
 - This handout's sequel is largely based on the "Basic Concepts" section of this wiki book

Resources

- **F# Software Foundation** <http://fsharp.org/>
- **Visual F#** on MSDN <https://msdn.microsoft.com/en-us/library/dd233154.aspx>
- **F# for Fun and Profit**
<http://fsharpforfunandprofit.com>
- **Try F#** <http://www./tryfsharp.org>
- **F# Programming**
http://en.wikibooks.org/wiki/F_Sharp_Programming
 - This handout's sequel is largely based on the “**Basic Concepts**” section of this wiki book

Resources — Sources

- **The Open Edition of the F# compiler, core library and tools** <https://github.com/fsharp/fsharp>
- `src/fsharp/FSharp.Core` <https://github.com/fsharp/fsharp/tree/master/src/fsharp/FSharp.Core>
- `control.fs`: `async` (async workflows), `mailbox` (agents/actors), ...
- `list.fs`: `List` module
- `seq.fs`: `Seq` module
- `prim-type.fs`: operators, comparisons, lists (type), conversions, ...

Resources — Sources

- **The Open Edition of the F# compiler, core library and tools** <https://github.com/fsharp/fsharp>
- **src/fsharp/FSharp.Core** <https://github.com/fsharp/fsharp/tree/master/src/fsharp/FSharp.Core>
- **control.fs**: **async** (async workflows), **mailbox** (agents/actors), ...
- **list.fs**: **List** module
- **seq.fs**: **Seq** module
- **prim-type.fs**: operators, comparisons, lists (type), conversions, ...

Resources — Sources

- **The Open Edition of the F# compiler, core library and tools** <https://github.com/fsharp/fsharp>
- **src/fsharp/FSharp.Core** <https://github.com/fsharp/fsharp/tree/master/src/fsharp/FSharp.Core>
- **control.fs: async** (async workflows), **mailbox** (agents/actors), ...
- **list.fs: List** module
- **seq.fs: Seq** module
- **prim-type.fs**: operators, comparisons, lists (type), conversions, ...

Basics

- Syntax (a bit controversial)
 - light : based on **indentation** (cf., Python); *offside rules!*
 - verbose ☹
- Note
 - types are rarely explicit : powerful **type inference** (but very **strongly** typed – **no** automatic conversions)
 - by default, “variables” are **immutable values** (cf., const)
 - powerful **pattern matching** (kind of a more powerful switch case)
- Namespaces and modules, with
 - functions
 - statements
 - classes (objects)

Basics

- Syntax (a bit controversial)
 - light : based on **indentation** (cf., Python); *offside rules!*
 - verbose ☹️
- Note
 - types are rarely explicit : powerful **type inference** (but very **strongly** typed – **no** automatic conversions)
 - by default, “variables” are **immutable values** (cf., const)
 - powerful **pattern matching** (kind of a more powerful switch case)
- Namespaces and modules, with
 - functions
 - statements
 - classes (objects)

Basics

- Syntax (a bit controversial)
 - light : based on **indentation** (cf., Python); *offside rules!*
 - verbose ☹
- Note
 - types are rarely explicit : powerful **type inference** (but very **strongly** typed – **no** automatic conversions)
 - by default, “variables” are **immutable values** (cf., const)
 - powerful **pattern matching** (kind of a more powerful switch case)
- Namespaces and modules, with
 - functions
 - statements
 - classes (objects)

Basics

- Syntax (a bit controversial)
 - light : based on **indentation** (cf., Python); *offside rules!*
 - verbose ☹
- Note
 - types are rarely explicit : powerful **type inference** (but very **strongly** typed – **no** automatic conversions)
 - by default, “variables” are **immutable values** (cf., const)
 - powerful **pattern matching** (kind of a more powerful switch case)
- Namespaces and modules, with
 - functions
 - statements
 - classes (objects)

Basics

- Syntax (a bit controversial)
 - light : based on **indentation** (cf., Python); *offside rules!*
 - verbose ☹
- Note
 - types are rarely explicit : powerful **type inference** (but very **strongly** typed – **no** automatic conversions)
 - by default, “variables” are **immutable values** (cf., const)
 - powerful **pattern matching** (kind of a more powerful switch case)
- Namespaces and modules, with
 - functions
 - statements
 - classes (objects)

Starting

- Light syntax based on indentation - **default**

```
1 #light
```

- using System

```
1 open System
```

- y is an **immutable value** of F# type **float** (C# **double** !)

```
1 let y = 3.0
```

- F# **printf** is inspired from C

```
1 printfn "y = %g" y // y = 3
```

- Can also use C# API

```
1 Console.WriteLine("y = {0}", y) // y = 3
```

Starting

- Light syntax based on indentation - **default**

```
1 #light
```

- using System

```
1 open System
```

- y is an **immutable value** of F# type **float** (C# **double** !)

```
1 let y = 3.0
```

- F# **printf** is inspired from C

```
1 printfn "y = %g" y // y = 3
```

- Can also use C# API

```
1 Console.WriteLine("y = {0}", y) // y = 3
```

Starting

- Light syntax based on indentation - **default**

```
1 #light
```

- using System

```
1 open System
```

- y is an **immutable value** of F# type **float** (C# **double !**)

```
1 let y = 3.0
```

- F# **printf** is inspired from C

```
1 printfn "y = %g" y // y = 3
```

- Can also use C# API

```
1 Console.WriteLine("y = {0}", y) // y = 3
```

Starting

- Light syntax based on indentation - **default**

```
1 #light
```

- using System

```
1 open System
```

- y is an **immutable value** of F# type **float** (C# **double** !)

```
1 let y = 3.0
```

- F# **printf** is inspired from C

```
1 printfn "y = %g" y // y = 3
```

- Can also use C# API

```
1 Console.WriteLine("y = {0}", y) // y = 3
```

Starting

- Light syntax based on indentation - **default**

```
1 #light
```

- using System

```
1 open System
```

- y is an **immutable value** of F# type **float** (C# **double** !)

```
1 let y = 3.0
```

- F# **printf** is inspired from C

```
1 printfn "y = %g" y // y = 3
```

- Can also use C# API

```
1 Console.WriteLine("y = {0}", y) // y = 3
```

Complete simple example F#

```
1 module Simple
2 open System
3 (* This can be a multi-line comment *)
4 // This is a single-line comment
5
6 let rec fib n =
7     match n with
8         | 0 -> 1
9         | 1 -> 1
10        | _ -> fib (n - 1) + fib (n - 2)
11
12 [<EntryPoint>]
13 let main args =
14     Console.WriteLine("fib 5: {0}", (fib 5))
15     0
```

- these are static functions (cp. instance/virtual)

Same simple example translated into C#

```
1 using System;
2 public static class Simple {
3     public static int fib(int n) {
4         switch (n) {
5             case 0: return 1;
6             case 1: return 1;
7             default:
8                 return Simple.fib(n - 1) + Simple.fib(n - 2);
9         }
10    }
11    public static void main(string[] arg) {
12        Console.WriteLine("fib 5: {0}", Simple.fib(5));
13    }
14    public static void Main(string[] args) {
15        Simple.main(args);
16    }
17 }
```

Outline

- 1 Background
- 2 Overview
- 3 Type System and Type Inference**
- 4 Functions
- 5 Lambdas
- 6 printfn
- 7 Immutable Values and Mutable Variables
- 8 Pattern Matching
- 9 Collections
- 10 Recursion or Loops?
- 11 Exercise
- 12 Infinite virtual sequences

Data Types

- *primitives*: **int**, **single** ≠ **float** = **double**, **string**, ...
- extensions: **bigint** = **BigInteger**, **complex**, ...
- *functions*, of course, as first-class citizens; all are **curried** (i.e., functions of **one** parameter only)!
- *collections*: **seq**<'T> (= **IEnumerable**<T>), **array**<'T>, **list**<'T> (**immutable**, and $\approx \neq$ **List**<T>)
- *tuples* and *records* (\approx **struct**)
- *discriminated unions* (implemented as a .NET base class with tags + derived classes)
- other *types*: classes, interfaces

Data Types

- *primitives*: **int**, **single** ≠ **float** = **double**, **string**, ...
- extensions: **bigint** = **BigInteger**, **complex**, ...
- *functions*, of course, as first-class citizens; all are **curried** (i.e., functions of **one** parameter only)!
- *collections*: **seq**<'T> (= **IEnumerable**<T>), **array**<'T>, **list**<'T> (**immutable**, and $\approx \neq$ **List**<T>)
- *tuples* and *records* (\approx **struct**)
- *discriminated unions* (implemented as a .NET base class with tags + derived classes)
- other *types*: classes, interfaces

Data Types

- *primitives*: **int**, **single** ≠ **float** = **double**, **string**, ...
- extensions: **bigint** = **BigInteger**, **complex**, ...
- *functions*, of course, as first-class citizens; all are **curried** (i.e., functions of **one** parameter only)!
- *collections*: **seq**<'T> (= **IEnumerable**<T>), **array**<'T>, **list**<'T> (**immutable**, and $\approx \neq$ **List**<T>)
- *tuples* and *records* (\approx **struct**)
- *discriminated unions* (implemented as a .NET base class with tags + derived classes)
- other *types*: classes, interfaces

Data Types

- *primitives*: **int**, **single** ≠ **float** = **double**, **string**, ...
- extensions: **bigint** = **BigInteger**, **complex**, ...
- *functions*, of course, as first-class citizens; all are **curried** (i.e., functions of **one** parameter only)!
- *collections*: **seq**<'T> (= **IEnumerable**<T>), **array**<'T>, **list**<'T> (**immutable**, and $\approx \neq$ **List**<T>)
- *tuples* and *records* (\approx **struct**)
- *discriminated unions* (implemented as a .NET base class with tags + derived classes)
- other *types*: classes, interfaces

Data Types

- *primitives*: **int**, **single** ≠ **float** = **double**, **string**, ...
- extensions: **bigint** = **BigInteger**, **complex**, ...
- *functions*, of course, as first-class citizens; all are **curried** (i.e., functions of **one** parameter only)!
- *collections*: **seq**<'T> (= **IEnumerable**<T>), **array**<'T>, **list**<'T> (**immutable**, and $\approx \neq$ **List**<T>)
- *tuples* and *records* (\approx **struct**)
- *discriminated unions* (implemented as a .NET base class with tags + derived classes)
- other *types*: classes, interfaces

Data Types

- *primitives*: **int**, **single** ≠ **float** = **double**, **string**, ...
- extensions: **bigint** = **BigInteger**, **complex**, ...
- *functions*, of course, as first-class citizens; all are **curried** (i.e., functions of **one** parameter only)!
- *collections*: **seq**<'T> (= **IEnumerable**<T>), **array**<'T>, **list**<'T> (**immutable**, and $\approx \neq$ **List**<T>)
- *tuples* and *records* (\approx **struct**)
- *discriminated unions* (implemented as a .NET base class with tags + derived classes)
- *other types*: classes, interfaces

Data Types

- *primitives*: **int**, **single** ≠ **float** = **double**, **string**, ...
- extensions: **bigint** = **BigInteger**, **complex**, ...
- *functions*, of course, as first-class citizens; all are **curried** (i.e., functions of **one** parameter only)!
- *collections*: **seq**<'T> (= **IEnumerable**<T>), **array**<'T>, **list**<'T> (**immutable**, and $\approx \neq$ **List**<T>)
- *tuples* and *records* (\approx **struct**)
- *discriminated unions* (implemented as a .NET base class with tags + derived classes)
- other *types*: classes, interfaces

Bind values and print them

```
1 let x1: int = 3 + 4 // explicitly declared as int
2
3 let x2 = 3 + 4 // inferred as int, from right-hand-side
4
5 printfn "results = %d %d" x1 x2
```

- x1 and x2 are **immutable** values, improperly called “variables”
- 3 and 4 are int literals
- %d is an int format specification

Outline

- 1 Background
- 2 Overview
- 3 Type System and Type Inference
- 4 Functions**
- 5 Lambdas
- 6 printfn
- 7 Immutable Values and Mutable Variables
- 8 Pattern Matching
- 9 Collections
- 10 Recursion or Loops?
- 11 Exercise
- 12 Infinite virtual sequences

Simple unary functions on primitive types

```
1  let inc (x: int): int = x + 1 // explicitly declared
2
3  // val inc : int -> int      : Func<int , int>
4
5  printfn "inc 3 = %d" (inc 3)
6
7  let inci x = x + 1          // inferred types
8
9  // val inci : int -> int
10
11 printfn "inci 3 = %d" (inci 3)
```

- 1 is int \Rightarrow x is int \Rightarrow + is int addition \Rightarrow x+1 is int (no automatic conversion)
- parentheses (inc 3) are required, otherwise we would have *two* arguments to print

Curried function of “two parameters”

```
1 let addi x y = x + y
2
3 // val addi: int -> int -> int // int -> (int->int)
4 // Func<int , Func<int , int>>
5
6 printfn "addi 3 4 = %d" (addi 3 4)
```

- addi seems to be a function taking *two* int parameters
- In reality, addi is a **high-order function** (aka **functional**), taking **one** int parameter and returning another function taking **one** more int parameter
- The \rightarrow arrow is **right associative**: $\text{int} \rightarrow (\text{int} \rightarrow \text{int})$
- The invocation (addi 3 4) can also be written as ((addi 3) 4)
- Although overloaded, arithmetic operations (such as +) are *defaulted* as int operations, unless *explicitly* required otherwise

Curried function of “two parameters”

```
1 let addi x y = x + y
2
3 // val addi: int -> int -> int // int -> (int->int)
4 // Func<int , Func<int , int>>
5
6 printfn "addi 3 4 = %d" (addi 3 4)
```

- addi seems to be a function taking *two* int parameters
- In reality, addi is a **high-order function** (aka **functional**), taking **one** int parameter and returning another function taking **one** more int parameter
- The \rightarrow arrow is **right associative**: $\text{int} \rightarrow (\text{int} \rightarrow \text{int})$
- The invocation (addi 3 4) can also be written as ((addi 3) 4)
- Although overloaded, arithmetic operations (such as +) are *defaulted* as int operations, unless *explicitly* required otherwise

Curried function of “two parameters”

```
1 let addi x y = x + y
2
3 // val addi: int -> int -> int // int -> (int->int)
4 // Func<int , Func<int , int>>
5
6 printfn "addi 3 4 = %d" (addi 3 4)
```

- addi seems to be a function taking *two* int parameters
- In reality, addi is a **high-order function** (aka **functional**), taking **one** int parameter and returning another function taking **one** more int parameter
- The \rightarrow arrow is **right associative**: **int** \rightarrow (**int** \rightarrow **int**)
- The invocation (addi 3 4) can also be written as ((addi 3) 4)
- Although overloaded, arithmetic operations (such as +) are *defaulted* as int operations, unless *explicitly* required otherwise

Curried function of “two parameters”

```
1 let addi x y = x + y
2
3 // val addi: int -> int -> int // int -> (int->int)
4 // Func<int , Func<int , int>>
5
6 printfn "addi 3 4 = %d" (addi 3 4)
```

- addi seems to be a function taking *two* int parameters
- In reality, addi is a **high-order function** (aka **functional**), taking **one** int parameter and returning another function taking **one** more int parameter
- The \rightarrow arrow is **right associative**: $\text{int} \rightarrow (\text{int} \rightarrow \text{int})$
- The invocation (addi 3 4) can also be written as ((addi 3) 4)
- Although overloaded, arithmetic operations (such as +) are *defaulted* as int operations, unless *explicitly* required otherwise

Curried function of “two parameters”

```
1 let addi x y = x + y
2
3 // val addi: int -> int -> int // int -> (int->int)
4 // Func<int , Func<int , int>>
5
6 printfn "addi 3 4 = %d" (addi 3 4)
```

- addi seems to be a function taking *two* int parameters
- In reality, addi is a **high-order function** (aka **functional**), taking **one** int parameter and returning another function taking **one** more int parameter
- The \rightarrow arrow is **right associative**: $\text{int} \rightarrow (\text{int} \rightarrow \text{int})$
- The invocation (addi 3 4) can also be written as ((addi 3) 4)
- Although overloaded, arithmetic operations (such as +) are *defaulted* as int operations, unless *explicitly* required otherwise

Partial evaluation

```
1 let addi3 = addi 3
2
3 // val addi3 : int -> int
4
5 printfn "addi3 4 = %d" (addi3 4)
6
7 printfn "addi3 7 = %d" (addi3 7)
```

- The expression (addi 3) is the **partial evaluation** of addi after fixing its first parameter a to 3
- The returned addi3 is a classical low-level function, expecting one more int parameter

Curried function of “two parameters” - op version

```
1 let (++) x y = x + y
2 // (++): int -> int -> int // int -> (int->int)
3
4 let a = (++) 3 4
5 let c = 3 ++ 4
6 printfn "a=%d, c=%d" a c
7
8 let b = ((++) 3) 4
9
10 let ``+3`` = (++) 3
11 // (+3): int -> int
12
13 let d = ``+3`` 4
14 printfn "b=%d, d=%d" b d
```

Pointers to functions

```
1 type FIII = int -> int -> int
2 //   Func<int , Func<int , int>>
3
4 let f : FIII = addi // explicit type
5 let f = addi      // inferred type
6
7 printfn "f 3 4 = %d" (f 3 4)
8
9 let addf (x: float) (y: float) = x + y
10 // let g : FIII = addf compilation error
```

- FIII is an alias for the type of all functions with the signature **int** -> (**int** -> **int**)
- f is compatible and can be bound to addi, but not to addf
- For interacting with .NET, use the special construct **delegate**

Pointers to functions

```
1 type FIII = int -> int -> int
2 //   Func<int, Func<int, int>>
3
4 let f : FIII = addi // explicit type
5 let f = addi      // inferred type
6
7 printfn "f 3 4 = %d" (f 3 4)
8
9 let addf (x: float) (y: float) = x + y
10 // let g : FIII = addf compilation error
```

- FIII is an alias for the type of all functions with the signature **int** -> (**int** -> **int**)
- f is compatible and can be bound to addi, but not to addf
- For interacting with .NET, use the special construct **delegate**

Outline

- ① Background
- ② Overview
- ③ Type System and Type Inference
- ④ Functions
- ⑤ **Lambdas**
- ⑥ printfn
- ⑦ Immutable Values and Mutable Variables
- ⑧ Pattern Matching
- ⑨ Collections
- ⑩ Recursion or Loops?
- ⑪ Exercise
- ⑫ Infinite virtual sequences

Lambdas

```
1 let f x = x + 1 // F#  
2  
3 let g = fun x -> x + 1 // F# lambda  
4  
5 Func<int , int> h = x => x + 1 // C# lambda
```

- Lambdas are introduced by the keyword **fun**

Outline

- 1 Background
- 2 Overview
- 3 Type System and Type Inference
- 4 Functions
- 5 Lambdas
- 6 printfn**
- 7 Immutable Values and Mutable Variables
- 8 Pattern Matching
- 9 Collections
- 10 Recursion or Loops?
- 11 Exercise
- 12 Infinite virtual sequences

(printfn "...") defines a function!

```
1 printfn "two integers = %d %d" 4 7
2
3 (printfn "two integers = %d %d") 4 7
4
5 ((printfn "two integers = %d %d") 4) 7
6
7 // printfn "two integers = %d %d" 4.0 7.0
8 // compilation error
```

- The signature of (printfn *format*) is determined by the % format specifications which appear inside the format
- The compiler checks if the given % format specifications are **format compatible**
- E.g., %d is compatible with all integer types, %A with almost all objects

(printfn "...") defines a function!

```
1 printfn "two integers = %d %d" 4 7
2
3 (printfn "two integers = %d %d") 4 7
4
5 ((printfn "two integers = %d %d") 4) 7
6
7 // printfn "two integers = %d %d" 4.0 7.0
8 // compilation error
```

- The signature of (printfn *format*) is determined by the % format specifications which appear inside the format
- The compiler checks if the given % format specifications are **format compatible**
- E.g., %d is compatible with all integer types, %A with almost all objects

(printfn "...") defines a function!

```
1 let pr = printfn "two integers = %d %d"
2
3 // val pr : (int -> int -> unit)
4
5 pr 4 7
```

- pr is a high-order functional which in the end prints two int numbers given one after the other
- roughly, type `unit` corresponds to `void`; in F#, it also has a single value, `()`

(printfn "...") defines a function!

```
1 let pr = printfn "two integers = %d %d"
2
3 // val pr : (int -> int -> unit)
4
5 pr 4 7
```

- pr is a high-order functional which in the end prints two int numbers given one after the other
- roughly, type **unit** corresponds to **void**; in F#, it also has a single value, ()

Outline

- ① Background
- ② Overview
- ③ Type System and Type Inference
- ④ Functions
- ⑤ Lambdas
- ⑥ printfn
- ⑦ Immutable Values and Mutable Variables**
- ⑧ Pattern Matching
- ⑨ Collections
- ⑩ Recursion or Loops?
- ⑪ Exercise
- ⑫ Infinite virtual sequences

Mutable variables

```
1 let mutable c = 1
2
3 printfn "%d" c // 1
4
5 c <- c + 1
6
7 printfn "%d" c // 2
```

- c is explicitly declared **mutable**
- it can be modified by the `<-` operator (**not** by `=`)
- **mutable** variables are **not really** accepted in **closures**
- i.e. mutables in closures are automatically transformed in **ref** references, which do work with **closures**

Mutable variables

```
1 let mutable c = 1
2
3 printfn "%d" c // 1
4
5 c <- c + 1
6
7 printfn "%d" c // 2
```

- c is explicitly declared **mutable**
- it can be modified by the <- operator (**not** by =)
- **mutable** variables are **not really** accepted in **closures**
- i.e. mutables in closures are automatically transformed in **ref** references, which do work with **closures**

Outline

- 1 Background
- 2 Overview
- 3 Type System and Type Inference
- 4 Functions
- 5 Lambdas
- 6 printfn
- 7 Immutable Values and Mutable Variables
- 8 Pattern Matching**
- 9 Collections
- 10 Recursion or Loops?
- 11 Exercise
- 12 Infinite virtual sequences

Propositional calculus—Syntax

```

1  type Prop =                               // instances
2  |   False                                 // False      ≡ false
3  |   True                                  // True         ≡ true
4  |   Not of Prop                          // Not(p)       ≡  $\neg p$ 
5  |   And of Prop * Prop                   // And(p, q)    ≡  $p \wedge q$ 
6  |   Or of Prop * Prop                    // Or(p, q)     ≡  $p \vee q$ 

```

- This defines a **discriminated union** family, which will be discussed in more detail later
- Syntactically, Prop instances correspond to formulas of a simple propositional calculus with two logical values (True and False)
- Logical operators appear as prefix functions

Propositional calculus—Syntax

```

1  type Prop =                               // instances
2  | False                                     // False      ≡ false
3  | True                                       // True        ≡ true
4  | Not of Prop                               // Not(p)     ≡  $\neg p$ 
5  | And of Prop * Prop                       // And(p, q)  ≡  $p \wedge q$ 
6  | Or of Prop * Prop                        // Or(p, q)   ≡  $p \vee q$ 

```

- This defines a **discriminated union** family, which will be discussed in more detail later
- Syntactically, Prop instances correspond to formulas of a simple propositional calculus with two logical values (True and False)
- Logical operators appear as prefix functions

Propositional calculus—Syntax

```

1  type Prop =                               // instances
2  | False                                     // False      ≡ false
3  | True                                       // True        ≡ true
4  | Not of Prop                               // Not(p)      ≡  $\neg p$ 
5  | And of Prop * Prop                       // And(p, q)   ≡  $p \wedge q$ 
6  | Or of Prop * Prop                        // Or(p, q)    ≡  $p \vee q$ 

```

- This defines a **discriminated union** family, which will be discussed in more detail later
- Syntactically, Prop instances correspond to formulas of a simple propositional calculus with two logical values (True and False)
- Logical operators appear as prefix functions

Propositional calculus—Samples

```
1 let shouldBeFalse = And(False , Not False)
2
3 let shouldBeTrue = Or(True , Not True)
4
5 let complexLogic =
6     And(And(True, Or(Not(True), True)),
7         Or(And(True, Not(True)), Not(True)) )
```

- What are their logical values?

Propositional calculus—Semantics

```
1 let rec eval x =  
2   match x with  
3     | False -> false  
4     | True -> true  
5     | Not(prop) -> not (eval prop)  
6     | And(prop1, prop2) -> (eval prop1)&&(eval prop2)  
7     | Or(prop1, prop2) -> (eval prop1)|| (eval prop2)
```

- eval defines the correct semantics
- `rec` indicates a **recursive** function (required)
- here, `match` corresponds to a `switch/case` statement

Propositional calculus—Semantics

```
1 let rec eval x =  
2   match x with  
3     | False -> false  
4     | True -> true  
5     | Not(prop) -> not (eval prop)  
6     | And(prop1, prop2) -> (eval prop1)&&(eval prop2)  
7     | Or(prop1, prop2) -> (eval prop1)|| (eval prop2)
```

- eval defines the correct semantics
- **rec** indicates a **recursive** function (required)
- here, **match** corresponds to a **switch/case** statement

Propositional calculus—Semantics

```
1 let rec eval x =  
2   match x with  
3     | False -> false  
4     | True -> true  
5     | Not(prop) -> not (eval prop)  
6     | And(prop1, prop2) -> (eval prop1)&&(eval prop2)  
7     | Or(prop1, prop2) -> (eval prop1)|| (eval prop2)
```

- eval defines the correct semantics
- **rec** indicates a **recursive** function (required)
- here, **match** corresponds to a **switch/case** statement

Propositional calculus—Sample evaluations

```
1 printfn "shouldBeFalse: %b" (eval shouldBeFalse)
2 // false
3
4 printfn "shouldBeTrue: %b" (eval shouldBeTrue)
5 // true
6
7 printfn "complexLogic: %b" (eval complexLogic)
8 // false
```

- %b is the format specification for bool values

Outline

- ① Background
- ② Overview
- ③ Type System and Type Inference
- ④ Functions
- ⑤ Lambdas
- ⑥ printfn
- ⑦ Immutable Values and Mutable Variables
- ⑧ Pattern Matching
- ⑨ Collections**
- ⑩ Recursion or Loops?
- ⑪ Exercise
- ⑫ Infinite virtual sequences

Collections – Bird's eye view

```
1 let mylist = [ 10; 20; 30 ]
2
3 let myarray = [| 10; 20; 30 |]
4 let v = myarray.[2] // 30
5
6 let myseq = seq { for i in 1..3 do yield i*10 }
```

- native F# lists are simple linked lists
- F# arrays are C# .NET arrays
- F# seqs are C# .NET sequences, i.e. `IEnumerable<>` (e.g. virtual)
- discriminated unions, tuples, records, structs, classes, ...

Processing collections by high-level functionals

Specialised modules:

- native F# lists : List module : map, collect, filter, length, ...
- F# arrays : Array module : map, collect, filter, length, ...
- F# seqs : Seq module : map, collect, filter, length, ...
- IEnumerable<>: LINQ : Select, SelectMany, Where, Count, ...

Why specialised modules?

- List module : functions take and return native **lists** - **eager**
- Array module : functions take and return **arrays** - **eager**
- Seq module : functions take and return **sequences** - **deferred/lazy**, as much as possible (cf. LINQ)

Processing collections by high-level functionals

Specialised modules:

- native F# lists : List module : map, collect, filter, length, ...
- F# arrays : Array module : map, collect, filter, length, ...
- F# seqs : Seq module : map, collect, filter, length, ...
- IEnumerable<>: LINQ : Select, SelectMany, Where, Count, ...

Why specialised modules?

- List module : functions take and return native **lists** - **eager**
- Array module : functions take and return **arrays** - **eager**
- Seq module : functions take and return **sequences** - **deferred/lazy**, as much as possible (cf. LINQ)

Processing collections by high-level functionals

Specialised modules:

- native F# lists : List module : map, collect, filter, length, ...
- F# arrays : Array module : map, collect, filter, length, ...
- F# seqs : Seq module : map, collect, filter, length, ...
- IEnumerable<>: LINQ : Select, SelectMany, Where, Count, ...

Why specialised modules?

- List module : functions take and return native **lists** - **eager**
- Array module : functions take and return **arrays** - **eager**
- Seq module : functions take and return **sequences** - **deferred/lazy**, as much as possible (cf. LINQ)

Processing collections by high-level functionals

Specialised modules:

- native F# lists : List module : map, collect, filter, length, ...
- F# arrays : Array module : map, collect, filter, length, ...
- F# seqs : Seq module : map, collect, filter, length, ...
- IEnumerable<>: LINQ : Select, SelectMany, Where, Count, ...

Why specialised modules?

- List module : functions take and return native **lists** - **eager**
- Array module : functions take and return **arrays** - **eager**
- Seq module : functions take and return **sequences** - **deferred/lazy**, as much as possible (cf. LINQ)

Processing collections by high-level functionals

Specialised modules:

- native F# lists : List module : map, collect, filter, length, ...
- F# arrays : Array module : map, collect, filter, length, ...
- F# seqs : Seq module : map, collect, filter, length, ...
- IEnumerable<>: LINQ : Select, SelectMany, Where, Count, ...

Why specialised modules?

- List module : functions take and return native **lists** - **eager**
- Array module : functions take and return **arrays** - **eager**
- Seq module : functions take and return **sequences** - **deferred/lazy**, as much as possible (cf. LINQ)

Processing collections by high-level functionals

Specialised modules:

- native F# lists : List module : map, collect, filter, length, ...
- F# arrays : Array module : map, collect, filter, length, ...
- F# seqs : Seq module : map, collect, filter, length, ...
- IEnumerable<>: LINQ : Select, SelectMany, Where, Count, ...

Why specialised modules?

- List module : functions take and return native **lists** - **eager**
- Array module : functions take and return **arrays** - **eager**
- Seq module : functions take and return **sequences** - **deferred/lazy**, as much as possible (cf. LINQ)

Processing collections by high-level functionals

Specialised modules:

- native F# lists : List module : map, collect, filter, length, ...
- F# arrays : Array module : map, collect, filter, length, ...
- F# seqs : Seq module : map, collect, filter, length, ...
- IEnumerable<>: LINQ : Select, SelectMany, Where, Count, ...

Why specialised modules?

- List module : functions take and return native **lists** - **eager**
- Array module : functions take and return **arrays** - **eager**
- Seq module : functions take and return **sequences** - **deferred/lazy**, as much as possible (cf. LINQ)

Processing collections by high-level functionals

Specialised modules:

- native F# lists : List module : map, collect, filter, length, ...
- F# arrays : Array module : map, collect, filter, length, ...
- F# seqs : Seq module : map, collect, filter, length, ...
- IEnumerable<>: LINQ : Select, SelectMany, Where, Count, ...

Why specialised modules?

- List module : functions take and return native **lists** - **eager**
- Array module : functions take and return **arrays** - **eager**
- Seq module : functions take and return **sequences** - **deferred/lazy**, as much as possible (cf. LINQ)

Bird's eye view on lists

```
1 let collection = [ 10; 20; 30 ] // 10 :: [20; 30]
2   // 10 :: 20 :: [30]
3   // 10 :: 20 :: 30 :: []
```

- Sidebar on lists: **singly linked** lists, head access is $O(1)$, element access is $O(n)$.

```
1 type list <'T> = // also 'T list
2   | ([])
3   | (::) of 'T * list <'T>
```

- `'T list` inherited from OCAML;
- `list <'T>` inherited from .NET
- For more details, please see the List module:
<http://msdn.microsoft.com/en-us/library/dd233224.aspx>

Bird's eye view on lists

```
1 let collection = [ 10; 20; 30 ] // 10 :: [20; 30]
2   // 10 :: 20 :: [30]
3   // 10 :: 20 :: 30 :: []
```

- Sidebar on lists: **singly linked** lists, head access is $O(1)$, element access is $O(n)$.

```
1 type list <'T> = // also 'T list
2   | ([])
3   | (::) of 'T * list <'T>
```

- `'T list` inherited from OCAML;
- `list <'T>` inherited from .NET
- For more details, please see the List module:
<http://msdn.microsoft.com/en-us/library/dd233224.aspx>

Bird's eye view on lists

```
1 let collection = [ 10; 20; 30 ] // 10 :: [20; 30]
2   // 10 :: 20 :: [30]
3   // 10 :: 20 :: 30 :: []
```

- Sidebar on lists: **singly linked** lists, head access is $O(1)$, element access is $O(n)$.

```
1 type list <'T> = // also 'T list
2   | ([])
3   | (::) of 'T * list <'T>
```

- `'T list` inherited from OCAML;
- `list <'T>` inherited from .NET
- For more details, please see the List module:
<http://msdn.microsoft.com/en-us/library/dd233224.aspx>

Bird's eye view on lists

```
1 let collection = [ 10; 20; 30 ] // 10 :: [20; 30]
2   // 10 :: 20 :: [30]
3   // 10 :: 20 :: 30 :: []
```

- Sidebar on lists: **singly linked** lists, head access is $O(1)$, element access is $O(n)$.

```
1 type list <'T> = // also 'T list
2   | ([])
3   | (::) of 'T * list <'T>
```

- `'T list` inherited from OCAML;
- `list <'T>` inherited from .NET
- For more details, please see the List module:
<http://msdn.microsoft.com/en-us/library/dd233224.aspx>

Bird's eye view on lists

```
1 let collection = [ 10; 20; 30 ] // 10 :: [20; 30]
2   // 10 :: 20 :: [30]
3   // 10 :: 20 :: 30 :: []
```

- Sidebar on lists: **singly linked** lists, head access is $O(1)$, element access is $O(n)$.

```
1 type list <'T> = // also 'T list
2   | ([])
3   | (::) of 'T * list <'T>
```

- `'T list` inherited from OCAML;
- `list <'T>` inherited from .NET
- For more details, please see the List module:
<http://msdn.microsoft.com/en-us/library/dd233224.aspx>

Bird's eye view on lists

```
1 let collection = [ 10; 20; 30 ] // 10 :: [20; 30]
2   // 10 :: 20 :: [30]
3   // 10 :: 20 :: 30 :: []
```

- Sidebar on lists: **singly linked** lists, head access is $O(1)$, element access is $O(n)$.

```
1 type list <'T> = // also 'T list
2   | ([])
3   | (::) of 'T * list <'T>
```

- `'T list` inherited from OCAML;
- `list <'T>` inherited from .NET
- For more details, please see the List module:
<http://msdn.microsoft.com/en-us/library/dd233224.aspx>

Collections: list intro

- our collection is an F# specific immutable literal list (\approx `new LinkedList<int> { 10, 20, 30 }`)
- is a non-empty list which has a head item, 10, and a tail list, `[20; 30]`
- could have also been constructed by pre-pending the head 10 to the list `[20; 30]`: `10 :: [20; 30]`, and so on...
- the empty list is indicated by `[]`
- operator `::` can be used in two ways:
 - 1 to *construct* lists (on the "right-hand side") and
 - 2 to *de-construct* lists, in **pattern matching** (on the "left-hand side", more later)

Collections: list intro

- our collection is an F# specific immutable literal list (\approx `new LinkedList<int> { 10, 20, 30 }`)
- is a non-empty list which has a head item, 10, and a tail list, `[20; 30]`
- could have also been constructed by pre-pending the head 10 to the list `[20; 30]`: `10 :: [20; 30]`, and so on...
- the empty list is indicated by `[]`
- operator `::` can be used in two ways:
 - 1 to *construct* lists (on the “right-hand side”) and
 - 2 to *de-construct* lists, in **pattern matching** (on the “left-hand side”, more later)

Collections: list intro

- our collection is an F# specific immutable literal list (\approx `new LinkedList<int> { 10, 20, 30 }`)
- is a non-empty list which has a head item, 10, and a tail list, `[20; 30]`
- could have also been constructed by pre-pending the head 10 to the list `[20; 30]`: `10 :: [20; 30]`, and so on...
- the empty list is indicated by `[]`
- operator `::` can be used in two ways:
 - ① to *construct* lists (on the “right-hand side”) and
 - ② to *de-construct* lists, in **pattern matching** (on the “left-hand side”, more later)

Collections: list intro

- our collection is an F# specific immutable literal list (\approx `new LinkedList<int> { 10, 20, 30 }`)
- is a non-empty list which has a head item, 10, and a tail list, `[20; 30]`
- could have also been constructed by pre-pending the head 10 to the list `[20; 30]`: `10 :: [20; 30]`, and so on...
- the empty list is indicated by `[]`
- operator `::` can be used in two ways:
 - ① to *construct* lists (on the “right-hand side”) and
 - ② to *de-construct* lists, in **pattern matching** (on the “left-hand side”, more later)

Collections: list intro

- our collection is an F# specific immutable literal list (\approx `new LinkedList<int> { 10, 20, 30 }`)
- is a non-empty list which has a head item, 10, and a tail list, `[20; 30]`
- could have also been constructed by pre-pending the head 10 to the list `[20; 30]`: `10 :: [20; 30]`, and so on...
- the empty list is indicated by `[]`
- operator `::` can be used in two ways:
 - ① to *construct* lists (on the “right-hand side”) and
 - ② to *de-construct* lists, in **pattern matching** (on the “left-hand side”, more later)

Outline

- ① Background
- ② Overview
- ③ Type System and Type Inference
- ④ Functions
- ⑤ Lambdas
- ⑥ printfn
- ⑦ Immutable Values and Mutable Variables
- ⑧ Pattern Matching
- ⑨ Collections
- ⑩ Recursion or Loops?
- ⑪ Exercise
- ⑫ Infinite virtual sequences

Processing collections

```
1 let x = [10; 20; 30]
2
3 // explicit loops
4 for v in x do printfn "%d" v
5
6 // high-level functionals
7 x |> List.iter (fun v -> printfn "%d" v)
8
9 List.iter (fun v -> printfn "%d" v) x
```

Recursive processing with pattern matching

```
1 // let rec print (x: int list) = // list<int>
2
3 let rec print x =
4     match x with
5     | [] -> ()
6     | v :: tail ->
7         printfn "%d" v
8         print tail
9
10 print x
```

- here, `::` is used for **pattern matching** and indicates an automatic decomposition of list items into a head and a tail

Various ways to map a collection

```
1 let c = [100; 200; 300]
2
3 let c2 = c |> List.map (fun x -> x+x)
4
5 let c2' = c |> Seq.map (fun x -> x+x)
6
7 let c2'' = c.Select(fun x -> x+x)
8
9 let c2''' = query { for x in c do select (x+x) }
10
11 printfn "%A, %A, %A, %A" c2 c2' c2'' c2'''
```

- Seq module \approx Enumerable, `seq<'T> = IEnumerable<T>`
- F#'s query expressions are similar to C#'s query expressions (comprehensions); more later

Various ways to map a collection

```
1 let c = [100; 200; 300]
2
3 let c2 = c |> List.map (fun x -> x+x)
4
5 let c2' = c |> Seq.map (fun x -> x+x)
6
7 let c2'' = c.Select(fun x -> x+x)
8
9 let c2''' = query { for x in c do select (x+x) }
10
11 printfn "%A, %A, %A, %A" c2 c2' c2'' c2'''
```

- Seq module \approx Enumerable, `seq<'T> = IEnumerable<T>`
- F#'s query expressions are similar to C#'s query expressions (comprehensions); more later

Various ways to map a collection

```
1 let c = [100; 200; 300]
2
3 let c2 = c |> List.map (fun x -> x+x)
4
5 let c2' = c |> Seq.map (fun x -> x+x)
6
7 let c2'' = c.Select(fun x -> x+x)
8
9 let c2''' = query { for x in c do select (x+x) }
10
11 printfn "%A, %A, %A, %A" c2 c2' c2'' c2'''
```

- Seq module \approx Enumerable, `seq<'T> = IEnumerable<T>`
- F#'s query expressions are similar to C#'s query expressions (comprehensions); more later

Outline

- ① Background
- ② Overview
- ③ Type System and Type Inference
- ④ Functions
- ⑤ Lambdas
- ⑥ printfn
- ⑦ Immutable Values and Mutable Variables
- ⑧ Pattern Matching
- ⑨ Collections
- ⑩ Recursion or Loops?
- ⑪ Exercise
- ⑫ Infinite virtual sequences

Exercise — essentially, a catamorphism

Compute the sum of squares of a given float list, using:

- Imperative programming explicit loops and mutable variables
- FP with recursive functions
- FP with one F# fold or one LINQ Aggregate
- FP with a query expression

Sample input:

```
1 let data = [10.; 20.; 30.; 40.]
```

Sample output:

```
1 sum = 3000.
```

Exercise — Imperative programming

```
1 let data = [ 1. ; 2. ; 3. ; 4. ]
2 // 1^2 + 2^2 + 3^2 + 4^2 = 30
3
4 let sumsquares1 (data: float list) : float =
5     let mutable a = 0.
6     for x in data do
7         a <- a + x * x
8     a // return
9
10 printfn "%.0f" (sumsquares1 data)
```

Exercise — recursive functions

```
1  let data = [ 1. ; 2. ; 3. ; 4. ]
2  // 1^2 + 2^2 + 3^2 + 4^2 = 30
3
4  let rec sumsquares2 (data: float list) : float =
5      match data with
6      | [] -> 0.
7      | h :: t ->
8          (sumsquares2 t) + h * h
9
10 printfn "%.0f" (sumsquares2 data)
```

Challenge: make this **tail-recursive**!

Exercise — List.fold or LINQ Aggregate

```
1 let data = [ 1. ; 2. ; 3. ; 4. ]
2 // 1^2 + 2^2 + 3^2 + 4^2 = 30
3
4 let sumsquares3 (data: float list) : float =
5     data |> List.fold (fun a x -> a + x * x) 0.
6
7 printfn "%.0f" (sumsquares3 data)
8
9 let sumsquares4 (data: float list) : float =
10    data.Aggregate(0., fun a x -> a + x * x)
11
12 printfn "%.0f" (sumsquares4 data)
```

Exercise — LINQ-like query expression

```
1 let data = [ 1. ; 2. ; 3. ; 4. ]
2 // 1^2 + 2^2 + 3^2 + 4^2 = 30
3
4 let sumsquares5 (data: float list) : float =
5     query { for x in data do sumBy (x * x)}
6
7 printfn "%.0f" (sumsquares5 data)
```

Outline

- ① Background
- ② Overview
- ③ Type System and Type Inference
- ④ Functions
- ⑤ Lambdas
- ⑥ printfn
- ⑦ Immutable Values and Mutable Variables
- ⑧ Pattern Matching
- ⑨ Collections
- ⑩ Recursion or Loops?
- ⑪ Exercise
- ⑫ Infinite virtual sequences

Infinite virtual sequences

```
1 let pow2 = seq {  
2     let mutable n, n2 = 0, 1  
3     while true do  
4         yield n2  
5         n2 <- n2 * 2  
6         n <- n + 1  
7     }
```

```
1 pow2 |> Seq.take 20  
2     |> Seq.iteri (fun i k -> ...)
```

Infinite virtual sequences

```
1 let pow2 = seq {  
2   let mutable n, n2 = 0, 1  
3   while true do  
4     yield n2  
5     n2 <- n2 * 2  
6     n <- n + 1  
7   }
```

```
1 pow2 |> Seq.take 20  
2     |> Seq.iteri (fun i k -> ...)
```